第一章 TCP/IP简介

OSI模型与TCP/IP层次结构

OSI：

> 应用层 表示层 会话层
> 运输层
> 网络层
> 数据链路层 物理层

TCP/IP：

> 应用层 TELNET FTP SMTP HTTP 直接为进程提供服务
> 传输层 TCP UDP 不同进程
> 网际层 IP ICMP 不同主机
> 网络接口层

API：作为应用层与传输层之间的接口 分隔用户进程与内核 使应用不需要关心通信细节

IP数据报格式

IPv4：长度除IP选项外160位，20字节

4位版本信息（4代表IPv4） 4位首部长度(必须为4字节整数倍，不含选项与填充时为20字节) 8位服务类型 16位总长度（最大65535字节，通常被限制为576字节）
16位标识 3位标志（最低位MF显示后面有无分片，中间位DF显示是否允许分片） 13位段偏移量（这三个字段用于控制IP数据报分片与重组）
8位寿命（TTL，以秒为单位的生存时间）8位协议（TCP为6，UDP为17，ICMP为1）16位校验和（保证完整性)
32位源IP地址，32位目的IP地址

IP地址划分，网段与子网掩码

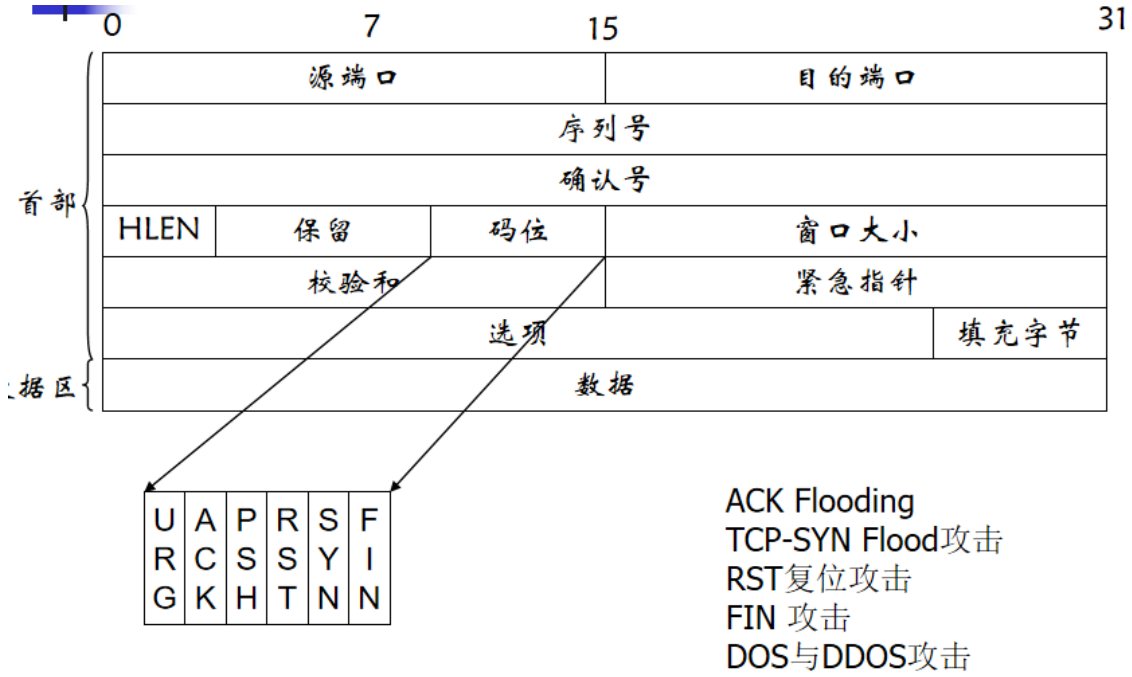
ABCDE类网络

> A类：二进制为0开头，网络号字段为1字节长，点分十进制开头范围为0-127
> B类：二进制10开头，网络号字段2字节长，开头范围为128-191
> C类：二进制110开头，网络字段号3字节长，开头范围为192-223

子网掩码

二进制以连续的1与连续的0组成，1部分对应的为网络号，0部分对应的主机号
与运算获得网络号
没有进行子网划分时，子网掩码长度默认与ABC类网络相同

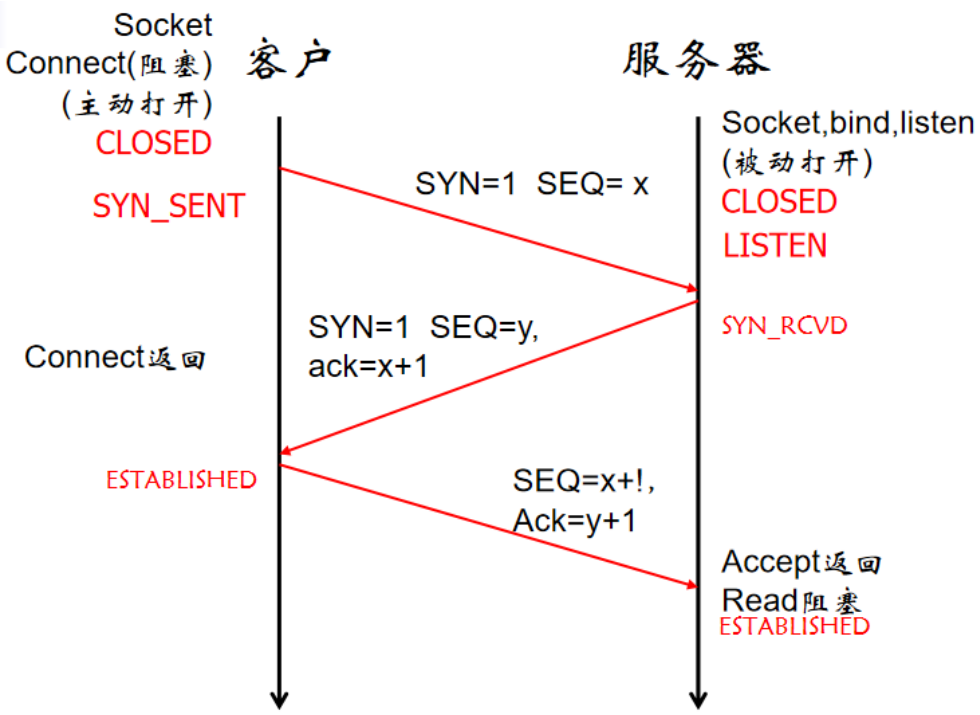
TCP报文格式



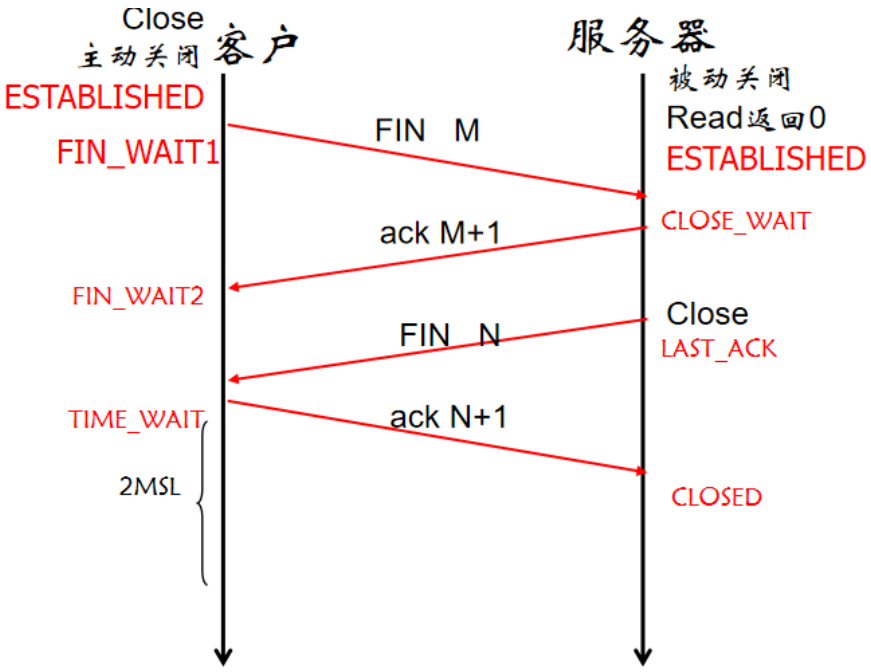
16位源端口 16位目的端口
32位序号（表示报文段中第一个字节的序号，用于排序与计数）
32位确认号（期望收到的数据的第一个字节的序号）
4位首部长度（4字节为单位，无选项情况下为20字节）6位保留 6位码元比特（6个标志，包含ACK，PSH，RST，SYN等）16位窗口（期望接受数据的长度）
16位校验和 16位紧急指针

三次握手四次挥手

三次握手

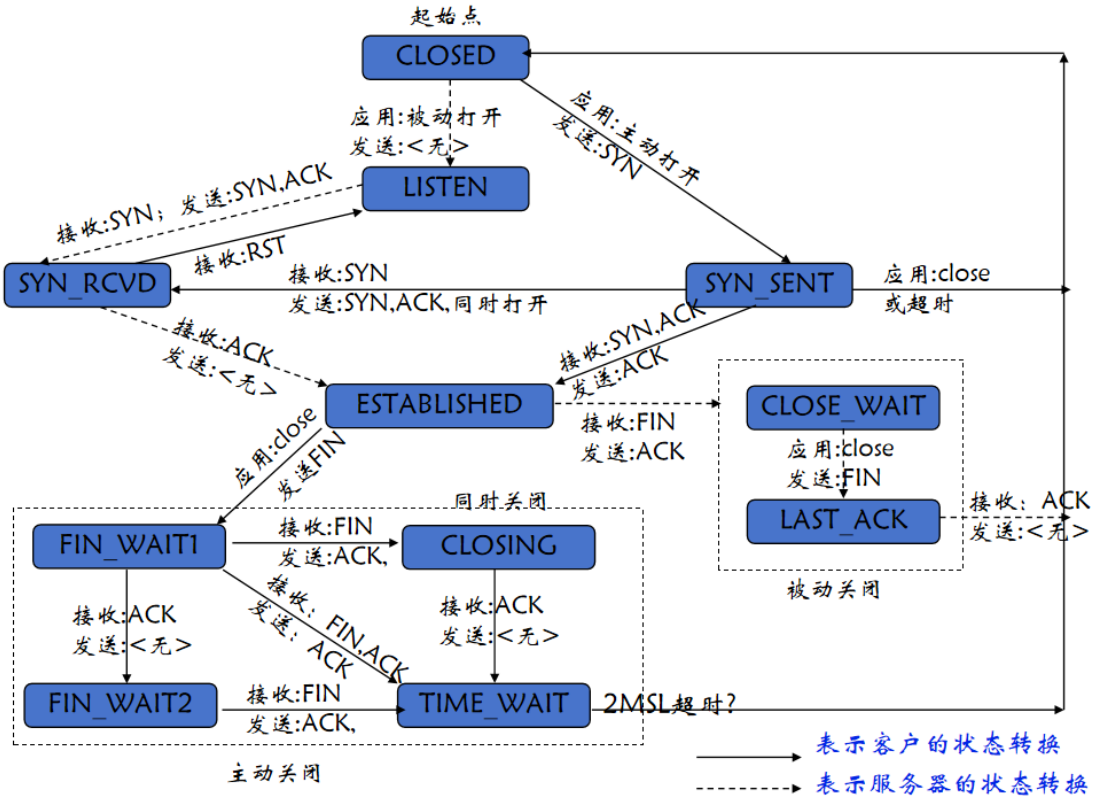


四次挥手



结合之后的内容，注意使用的是什么函数。

状态转换图



端口号

0-1023：IANA控制
1024-49151：IANA登记
49152-65535：临时端口

课后题

1.当TCP两端都关闭了链接，为什么主动关闭的一方还要在time_wait等待一段时间才能删除连接记录，并返回close状态?

2.从服务器进程的角度分析TCP建立和关闭连接时状态变化过程。

第二章 套接字编程简介

两种套接字

套接字接口(socket,又称为berkeley套接口)
传输层接口（TLI）

套接字是一种网络API

协议与套接字类型

IP协议：

> INET:IPv4
> INET6:IPv6

套接字类型：

> SOCK_STREAM 流式套接字 TCP协议 可靠
> SOCK_DGRAM 数据报套接字 无连接 UDP协议
> SOCK_RAW 原始套接字 检测新的协议

套接字地址结构

以sockaddr_in为例，这是IPv4的套接字地址结构（IPv6的为sockaddr_in6）
其中三个成员比较重要

> sin_family:Internet地址族，在IPv4为AF_INET（宏）
> sin_port:端口号，网络字节序
> sin_addr：结构体 其中的成员存储IP地址的值

因此有两种访问IP的方式：访问【saddr.sin_addr】与【saddr.sin_addr.s_addr】，前者访问结构，后者访问值，两者处理方式不同

通用套接字地址结构

结构名称为 sockaddr
具体的套接字地址结构用于对数据进行处理，通用套接字地址结构用于传参

字节排序（认识，用途）htons端口

hton ntoh：主机->网络 网络->主机
s l ：短（16位） 长（32位）
上下两两组合有四个函数，用于主机字节序与网络字节序之间的相互转换
htons往往用于端口号的字节排序

字节操纵：用途

由于套接字地址结构的IP字段包含多个字节的0，不是C字符串，对其操作需要使用特别的函数

```
void bzero(void *dest,size_t len)//用于将dest指向结构的len个字节置为0
void *memset(void *dest,int x,size_t len)//用于将dest指向结构的len个字节置为x

void bcopy(const void *src,void *dest,size_t len)//将src对应的len字节数据拷贝到dest，当src与dest相同时可以正确处理
void *memcpy(const void *dest,void *src,size_t len)//将src对应的len字节数据拷贝到dest,与上一个函数相比交换了源/目的参数位置,当src与dest相同时结果未知

int bcmp(const void *src,void *dest,size_t len);//比较src与dest字符串，相同返回0，不相同返回非零
int memcmp(const void *str1,const void *str2,size_t len);//功能相同，str1所指字节＞str所指字节 返回 ＞零
```

ip地址转换：转换点分十进制与二进制 函数定义，函数参数，函数用途，函数特点 返回值

如

```
in_addr_t inet_addr(const char *str);//将点分十进制格式的str转化为二进制,255.255.255.255不可用，返回值为储存二进制的结构
int inet_aton(const char *str,struct in_addr *numstr);//功能相同，可以处理255.255.255.255,处理后的二进制由第二个参数返回。
char *inet_ntoa(struct in_addr inaddr);//将二进制inaddr 转化为点分十进制字符串，返回值为处理后的字符串
```

这几个都只能处理IPv4的
又如

```
int inet_pton(int family,const char *str,void * number);//将family类型的，储存在str中的点分十进制ip地址，转化为二进制，在numstr中返回。family即AF_INET或AF_INET6
const char *inet_ntop(int family,const void * numstr,char *str,size_t len)//进行相反操作，处理结构由str指针返回，需要提前分配空间，len为目标长度。
```

传递套接字地址结构的函数 哪些 特点 方式

bind connect sendto
accept recvfrom
前三个函数需要传入地址结构与长度，后两个参数需要传入地址结构长度，函数通过参数返回地址结构
参数的格式均为通用套接字地址结构的指针（struct sockaddr *）

课后题

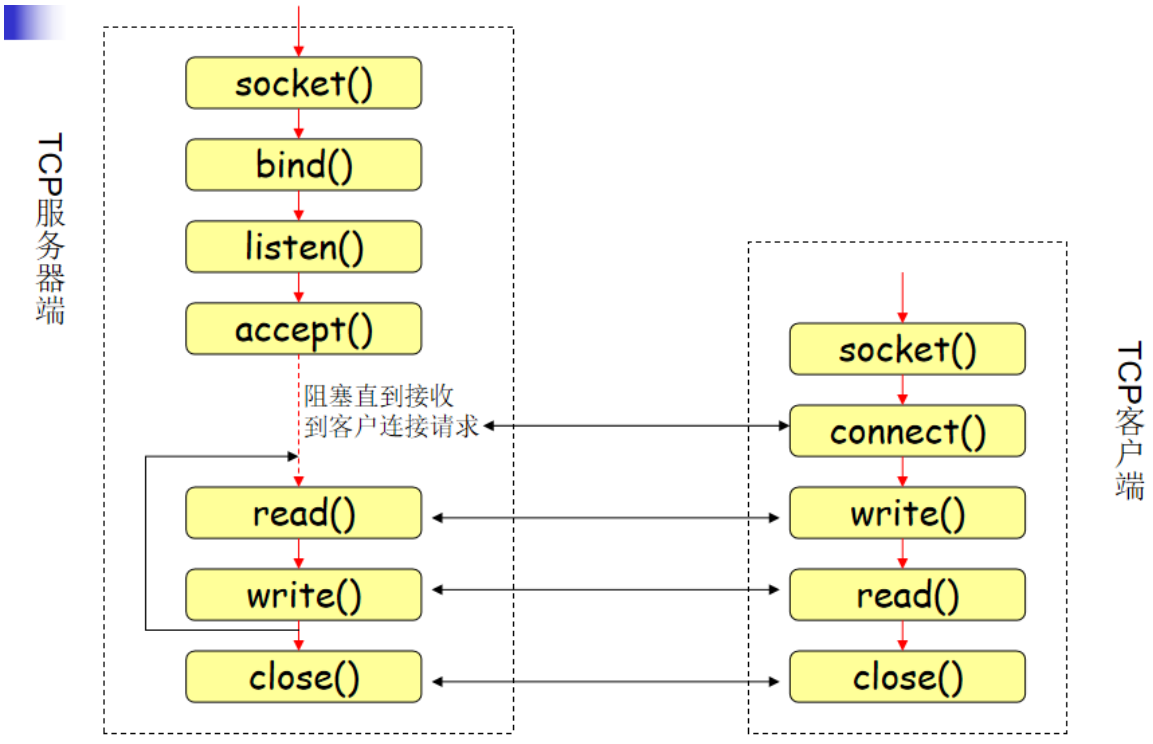
1.简述通用套接字地址结构的作用

2.简述字节排序函数的作用

3.举例说明从进程到内核和从内核到进程两种传递套接字地址结构的方式有什么区别。

第三章 基本TCP套接字编程

流程



socket函数 定义 参数 用途 返回值

```
int socket(int family,int type,int protocol);
//family指定协议族 AF_INET AF_INET6 AF_ROUTE(路由套接字)
//type指定套接字类型 SOCK_STREAM（字节流TCP） SOCK_DGRAM（数据包UDP） SOCK_RAW（原始套接字）
//protocol参数只在原始套接字类型中需要赋值，否则为0即可
```

该函数用于创建套接字描述符并返回，对应一个套接字，用于实现I/O操作
若创建失败，会返回-1

bind函数 同上 不调用的话

一般是服务端使用，客户端视情况也可以使用。若是不调用，内核将自动为套接字分配地址与端口
调用后为套接字指定 端口号和IP地址，或其中一个。

```
int bind(int sockfd,const struct sockaddr* server,socklen_len addrlen);
//sockfd socket函数返回值
//server 赋值后的地址结构，细节见后
//addrlen 地址结构的长度
```

> 注意：
>
> 参数addr中的相关字段在初始化时，必须是网络字节序；
>
> 如果由内核来选择IP地址和临时端口号，函数并不返回所选择的值。
>
> 为了获得这些值，进程必须调用getsockname函数

```
getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *localaddrlen);
```

常见错误：EAADRINUSE 表示地址已使用，通过设置套接字选项SO_REUSEADDR解决。

套接字选项设置

设置套接字的选项(属性)

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
//一般只有一种用法，见下：
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
//fd指向一个打开的套接字描述字
//opt通过如下方式定义
    int opt = 1
```

套接字地址结构的赋值

```
struct sockaddr_in server;
bzero(&server,sizeof(server));
server.sin_family=AF_INET;
server.sin_port=htons(【端口号】);//若为0则由内核选择
server.sin_addr.s_addr=inet_addr("【ip】");//若为INADDR_ANY则由内核选择
```

在调用需要使用地址的函数之前进行初始化

listen函数 同上

用于转换套接字为监听模式

```
int listen(int sockfd,int backlog);
//sockfd 经过该函数后，这个套接字将会转换为监听套接字，进入监听模式
//backlog 指定请求队列的最大链接个数，
```

对于一个监听套接字，内核将维护两个队列，一个为未连接，一个为已连接。
当收到SYN报文时，将会在未完成队列中创建新的条目，并开始回应客户端的握手过程。
当握手流程完成后，将会将该条目移到已完成队列。

connect函数 同上

这是一个客户端的函数，用于开始TCP三次握手，与远程服务器进行链接
执行成功后，socket参数对应的套接字会变成已连接套接字

```
int connect(int sockfd,const struct sockaddr *addr,socklen_t addrlen);
//sockfd参数，传入socket函数返回值
//addr参数，传入通用套接字地址结构格式的服务器地址
//addrlen 该地址结构的大小
```

accept函数 同上 注意addrlen参数

对应connect函数，用于接受客户端的连接请求
成功接受后，将返回一个新的套接字描述符，称为已连接套接字

```
int accept(int listenfd,struct sockaddr *client,sockklen_t *addrlen)
//listenfd listen函数返回的监听套接字
//client 用于返回连接对方的地址结构
//addrlen 既用来向函数传递地址的长度，也用来返回函数收到地址结构的长度
```

数据传输函数

write函数

发送数据，将返回数据的字节数。

```
int write(int sockfd,char *buf,int len);
//sockfd 已连接套接字
//buf 存储发送数据的缓冲区，可以是字符串，可以是申请的空间
//len 指明发送数据缓冲区的大小,一般可以用sizeof()，或者MAXDATASIZE这个宏
```

send函数

同样是数据发送，同write函数相比多了一个附加的参数，返回值含义也同write

```
ssize_T sent(int sockfd,const void *buf,size_t len,int flags);
//前三个参数与write相同
//最后一个是传输控制标志，为0则与write函数做相同操作
```

read函数

读取数据
返回值 > 0：读取成功
返回值=0：连接关闭
返回值 < 0：读取出错

```
int read(int sockfd,char *buf,int len);
//参数含义与使用与write大差不差
```

recv函数

recv函数之于read函数，如同send函数之于write函数，此处不细表。

close函数

用于关闭套接字，关闭之前会将待发送数据发送完
实质上是把套接字描述符的访问数-1，这点在多进程中体现的较为明显。
当访问计数=0时，才会触发TCP协议中的终止连接（四次挥手）操作

```
int close(int sockfd);
```

课后题

1.简述TCP套接字编程的基本步骤

2.简述bind函数中使用通配地址的作用

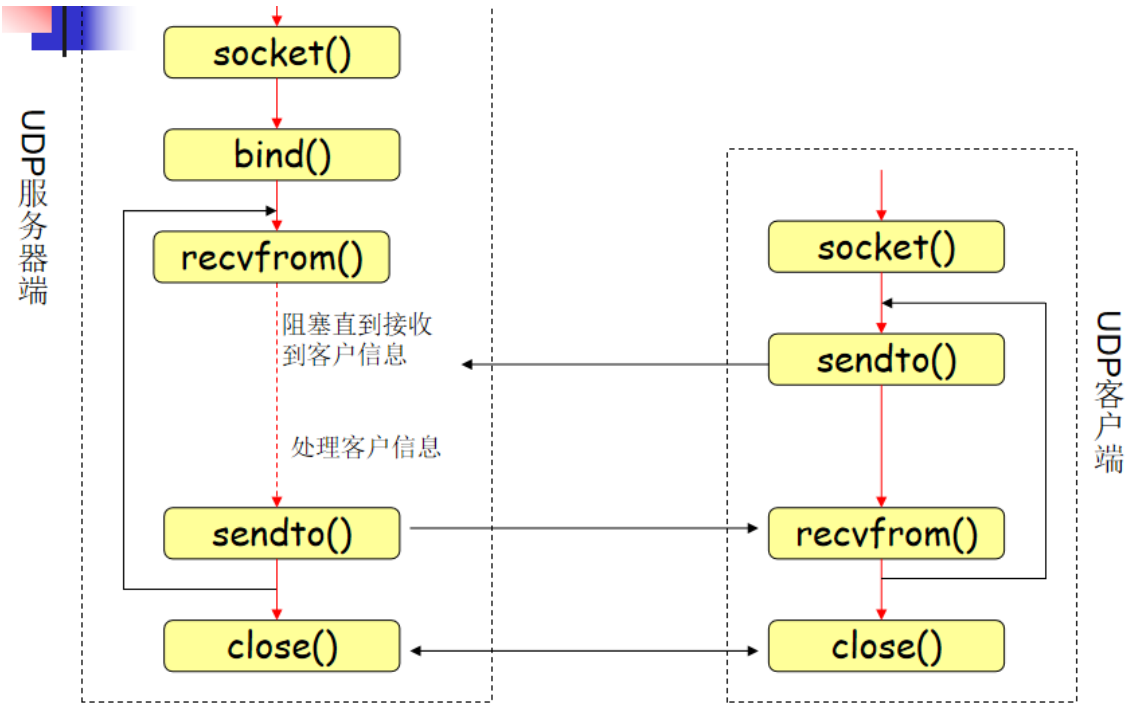
3.简述TCP套接字基本函数编程的过程中,TCP连接的状态变化。

第四章 基本UDP套接字编程

UDP实现的程序

DNS域名系统，NFS网络文件系统，SNMP简单网络管理协议

UDP流程



connect函数在UDP中的使用（场景：确定的唯一对方，异步错误）限制 优点

connect函数在UDP中使用后，会将对方的IP地址和端口号返回给进程，并改变参数所指套接字的性质
此时数据传输时只能使用sent，write，recv,read,而非sendto,recvfrom。
此时，套接字可以接收到异步错误。
　异步错误：客户端把UDP数据包发送给没用运行的服务器，服务器返回ICMP报文“端口不可达”，但recvfrom接受不到，客户端会永远阻塞于recvfrom函数。
一般用于UDP与确定的唯一对方进行通信。

recvfrom函数

即需要指定地址的recv

```
ssize_t recvfrom(int sockfd,void *buff,size_t len,int flags
                 struct sockaddr *from,size_t *addrlen);
//前四个参数与recv一致
//from参数 为通用套接字地址结构，用于返回消息来源的地址。
//addrlen 传入时指定from结构的长度，执行后为消息来源地址的长度。
//from参数可以为空指针，此时addrlen参数也必须是空指针，表示不关心数据方的协议地址。
```

sendto函数 0字节数据报

即需要指定地址的send

sendto可以发送长度为的数据报，由于UDP的无连接特性，sendto发送0 与 recvfrom返回0，均不意味着链接的关闭，这与TCP的read函数不同。

```
ssize_t sendto(int sockfd,void *buff,size_t len,int flags
                struct sockaddr *to,size_t *addrlen);
//前四个参数与send一致
//to参数 为通用套接字地址结构，用于指定消息来源的地址。
//addrlen 传入指定to结构的长度
```

响应验证

由于向UDP进程发送数据不需要建立链接，导致任何进程都有可能向客户发送数据，因此需要判断响应是否来自客户端

```
if(len !=  sizeof(server)
  ||memcmp((const void *)&server, (const void *)&peer, len) != 0)
    {
        printf("来自不是服务器的地方");
    }
else{
    printf("来自服务器");
}
```

课后题

1.简述UDP套接字编程的基本步骤

2.为什么要在recvfrom和sendto指明目的地址

3.简述connect函数在UDP套接字编程的作用

第五章 并发服务器

迭代与并发的区别

> 迭代：同时只能处理一个用户的请求
>
> 并发：同时可以处理多个用户的请求

并发的三种方式

> 多进程
>
> 多线程
>
> IO复用

进程并发

fork与vfork(子父进程)返回值，返回特点

fork与vfork函数均用于创建子进程

其区别在于：

> fork函数采用 Copy on write 的资源拷贝方式，在进程试图写数据之前，两个进程对数据共享。在进程写数据之后，会拷贝一份数据，以避免子父进程之间互相影响。

> fork函数执行后，若没有进行限制，则子父进程的执行顺序是不确定的。

> vfork函数采用共享的方式，子父进程共享使用数据，一个进程对数据的更改会影响到另一个数据的执行。
>
> vfork函数执行后，父进程会被阻塞，子进程借用父进程的地址空间运行，直到子进程退出或调用execve()函数。

调用fork函数或vfork函数成功后，会产生两个返回值。其中，子进程的返回值是0，而父进程的返回值是子进程的id号。

进程终止：僵尸进程 防范 wait waitpid函数

僵尸进程：已经终止，但父进程尚未对其进行善后处理

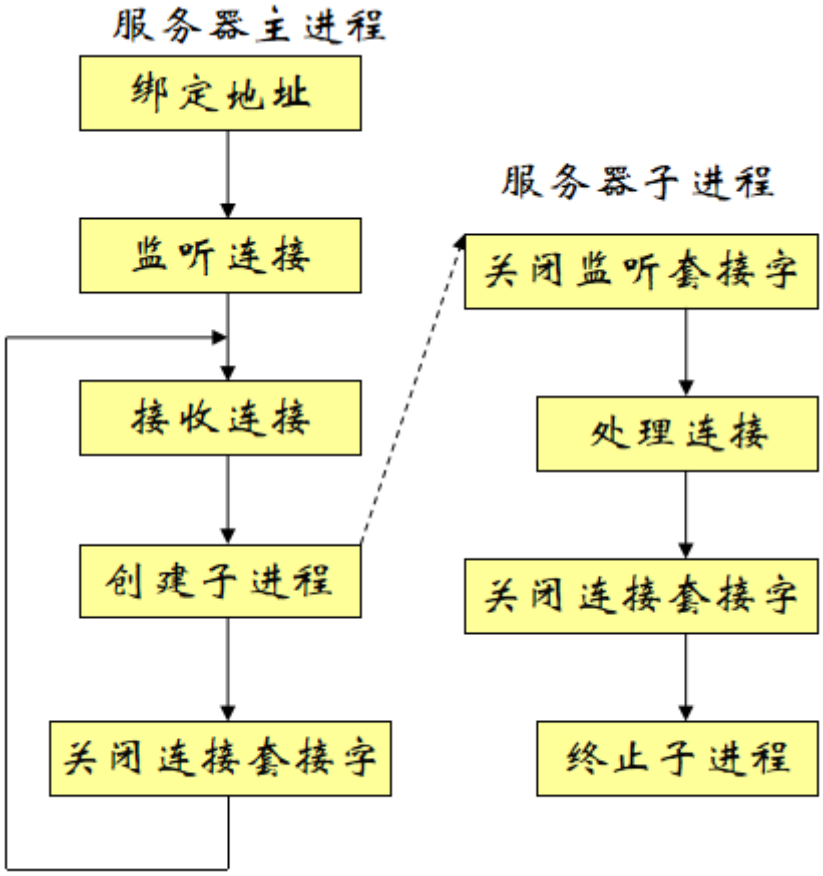
```
pid_t wait(int *statloc);  // 立即返回，并释放已终止子进程的资源。若没有已终止的子进程，但有正在执行的子进程，则会阻塞在这里。若没有子进程，则出错并返回-1
//成功执行会返回终止子进程的ID
//stat_loc参数返回子进程的终止状态。
```

若多个子进程同时终止，wait()函数可能只会执行一次，导致出现僵尸进程。

使用waitpid()函数可以通过更精确的控制来避免这个问题。

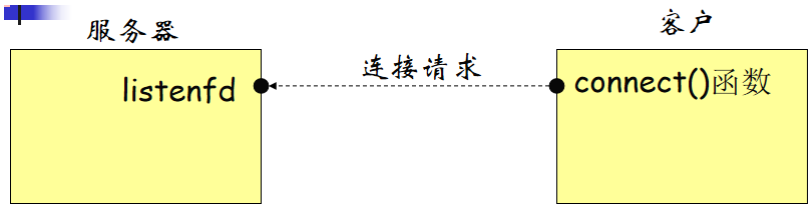
```
pid_t waitpid(pid_t pid,int *statloc,int option);
//pid指明关注哪些子进程。pid>0：关注进程号为pid的进程; pid<-1：关注进程组号为|pid|的所有进程; pid=-1：关注所有进程
//statloc参数与wait函数相同
//option参数指定附加选项，如WNO_HANG：内核中无已终止进程时不阻塞。
```

进程并发服务器流程

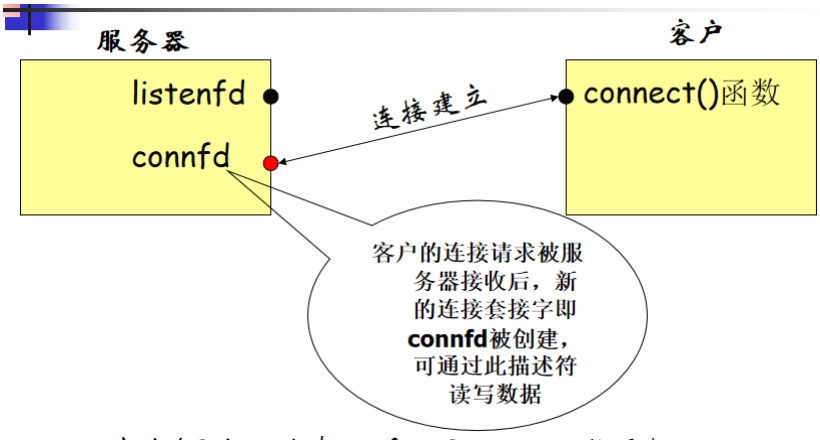


建立链接时，客户端与服务器的较为具体的流程如下：

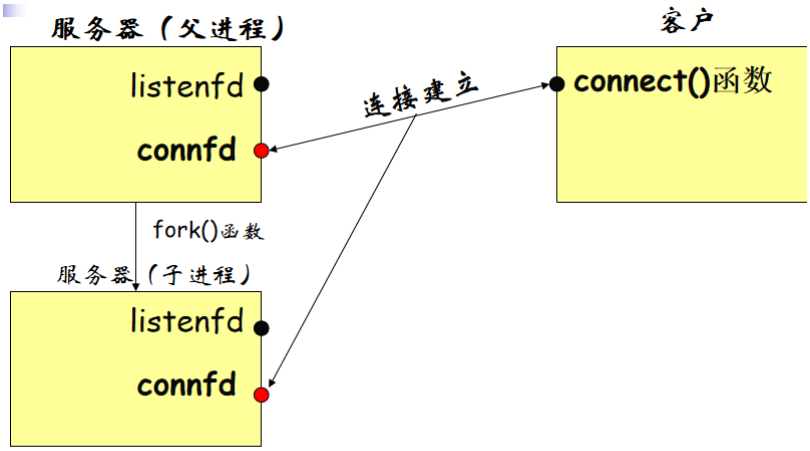
1.客户端向服务端发起链接请求



2.链接建立，服务端产生已连接套接字



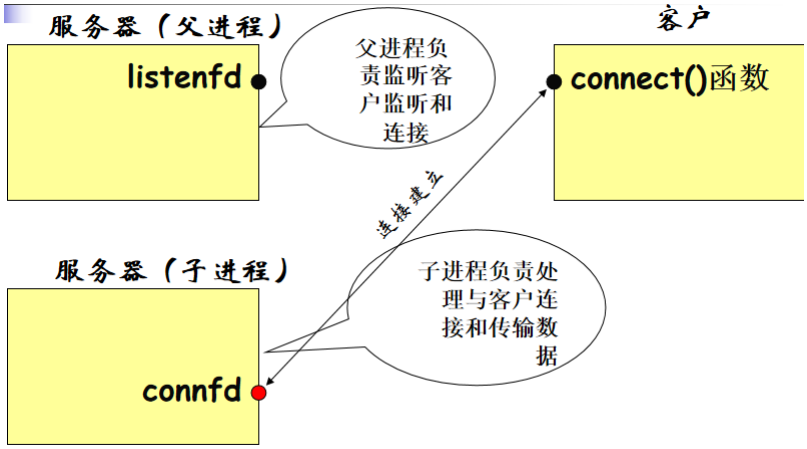
3.服务器调用fork函数，子父进程共享监听与已连接套接字



客户/服务器状态图（调用fork函数后）

4.父进程关闭已连接套接字，准备监听下一个连接请求

子进程关闭监听套接字，负责与建立连接的客户端通信



线程并发

pthread_create函数

用于创建新线程。

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *arg);
//tid：返回创建线程的id号
//attr：指向线程属性，通常设置为NULL
//func：线程函数，子线程将要执行的代码需要封装在一个函数中，称为线程函数。该函数的返回值必须是一个void类型的指针，它的参数也必须是有且仅有一个void类型的指针。
//arg：参数指针，当线程函数需要输入参数时，需要将参数打包成一个结构，然后通过arg指向这个结构，来进行传参。
```

执行该函数后，一个新线程将会被创建，并开始执行线程函数中的代码。

传递参数方式（3种）

1.通过强制类型转换，把要传递的数据直接转换为void *类型，传入后再转换回来

缺点：只能传递一个参数

2.主线程创建arg局部结构体变量，并给其赋值。用一个指针指向该结构并传入。

缺点：某个线程对arg的修改会影响到其他进程。

3.主线程为arg结构体分配空间并产生指针，将该指针传入。

pthread_join函数

等待线程终止，与waitpid函数功能类似。

```
int pthread_join(pthread_t tid, void **status);
```

等待的线程必须是当前进程的成员，且不能是分离的线程和守护线程。

pthread_detach函数

将某个线程变为分离的，分离的线程在终止后，会被系统释放拥有的所有资源。

```
int pthread_detach(pthread_t tid);
```

pthread_self函数

获取线程自己的ID号

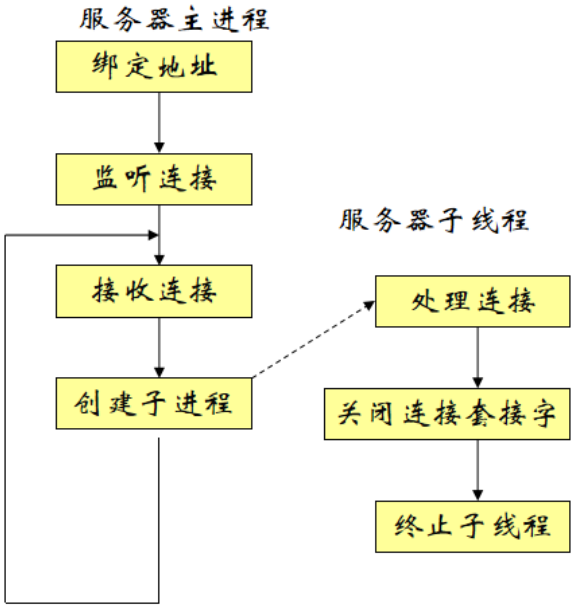
```
int pthread_self();
```

pthread_exit函数

终止当前线程

```
void pthread_exit();
```

线程流程



安全函数（线程安全问题的原因，可能出现问题的情况，避免）

由于线程之间内存空间共享，多个线程修改相同的内存会出现不可预料的后果。如静态变量。

TSD

即线程特定数据。

类似于全局变量，但是是线程私有的，是定义线程私有数据的唯一方法。

以关键字标志，通过函数进行读写。

pthread_key_create函数

创建一个TSD，以一个关键字进行标志。

```
int pthread_key_create(pthread_key_t *key,void(* destructor)(void *value));
//key 一般作为全局变量进行定义，如 pthread_key_t test_key;调用时传入其地址。
//此时传入该函数的参数即为&test_key，之后直接使用test_key作为参数进行读写
//destructor函数为可选的析构函数。在线程退出时执行，一般用于释放空间等。
```

每个进程只能调用一次。

pthread_once函数，

实现在进程中对某个函数只调用一次。

```
int pthread_once(pthread_once_t *once,void (*init)(void));
//函数使用once参数所指的变量，保证进程只执行一次init函数
//为实现这一功能，通常令once = PTHREAD_ONCE_INIT
```

pthread_setspecific函数与pthread_getspecific函数

实现对TSD的读写

```
int pthread_setspecific(pthread_key_t key,const void *value);
//key：之前定义的全局变量本身，如test_key
//value：通常动态分配的内存区域，如malloc返回的地址
//该函数此时可以理解为为线程关键字绑定一个已分配的空间，用于存储数据。
```

```
void * pthread_getspecific(pthread_key_t key);
//key：之前定义的全局变量本身，如test_key
//返回TSD绑定的值，在上面这种分配方式中，该函数会返回已分配空间的地址。
```

其他实现线程安全性的方式

使用函数参变量：将数据保存函数使用的索引 index 作为**调用【数据保存函数】的函数**的局部变量，平时保存在**调用函数**中，每次调用数据保存函数时需要传入。

调用函数需要为其分配空间并进行初始化。

课后题

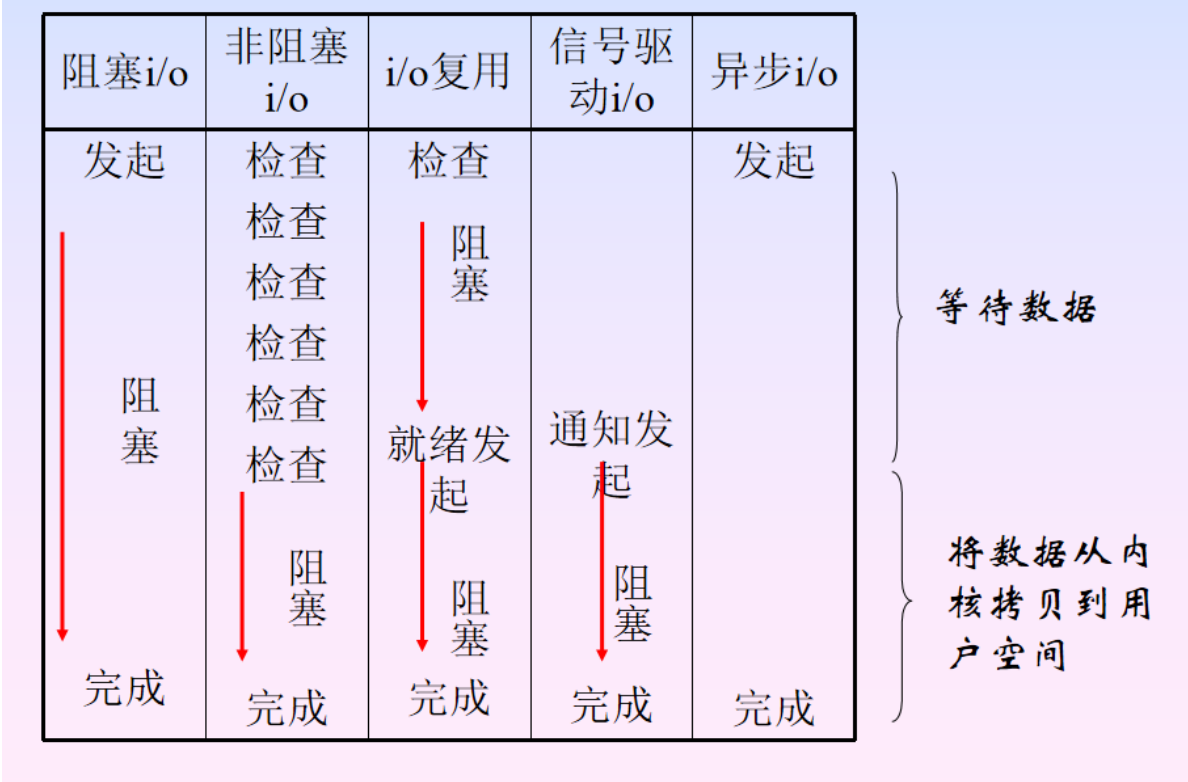
1.多进程和多线程设计上的区别

2.多进程并发服务器中，父进程创建子进程后，父子进程如何对描述符进行操作的，为什么。

3.比较wait和waitpid函数的区别

第六章 I/O编程

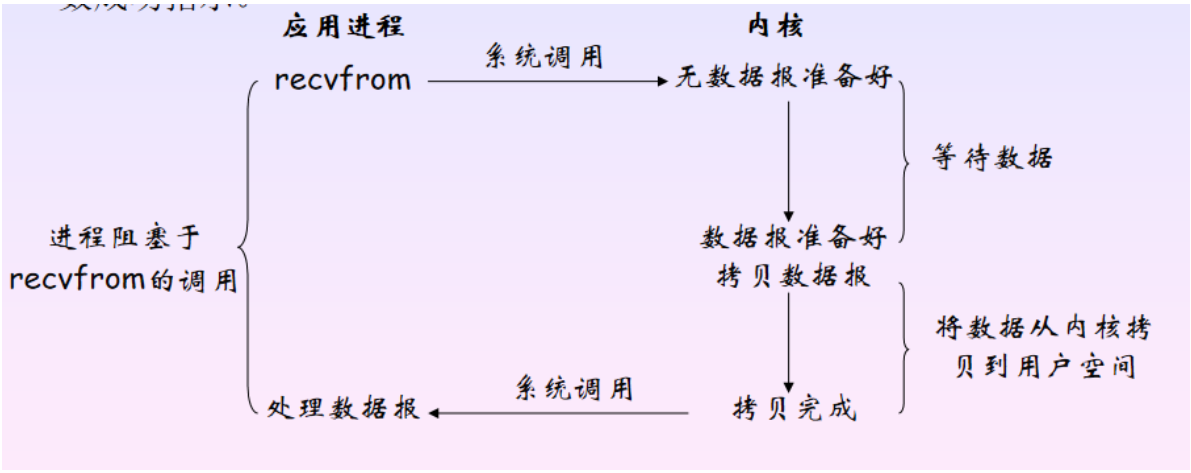
五个I/O模型的比较，区别



	阻塞I/O	非阻塞I/O	I/O复用	信号驱动I/O	异步I/O

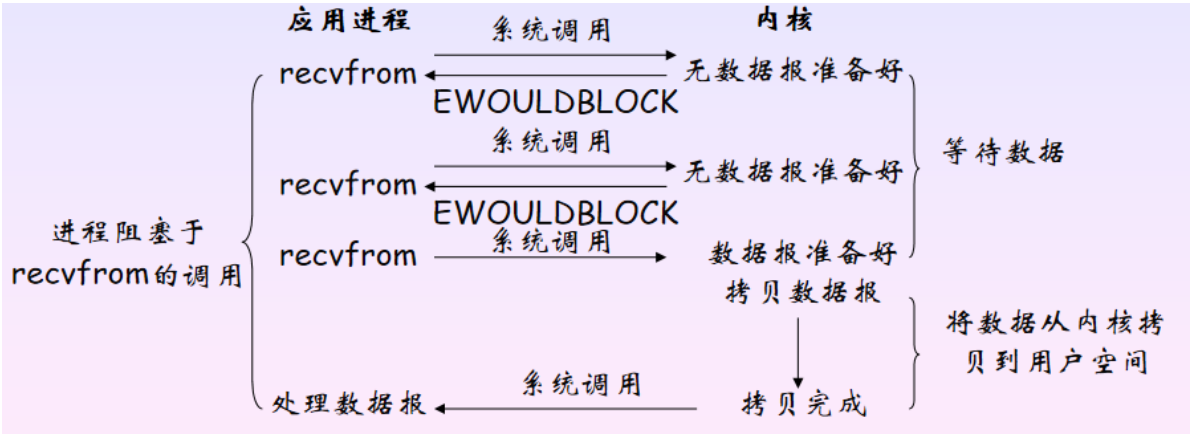
	阻塞I/O	非阻塞I/O	I/O复用	信号驱动I/O	异步I/O
等待过程是否阻塞	是，阻塞于recvfrom	否，使用轮询查询	是，阻塞于select或poll	否，等待完毕后通过信号通知，通过系统调用处理信号，系统调用立即返回	否
读取过程是否阻塞	是，阻塞于recvfrom	是，阻塞于recvfrom	是，阻塞于recvfrom	是，阻塞于recvfrom	否

阻塞I/O



最为流行。

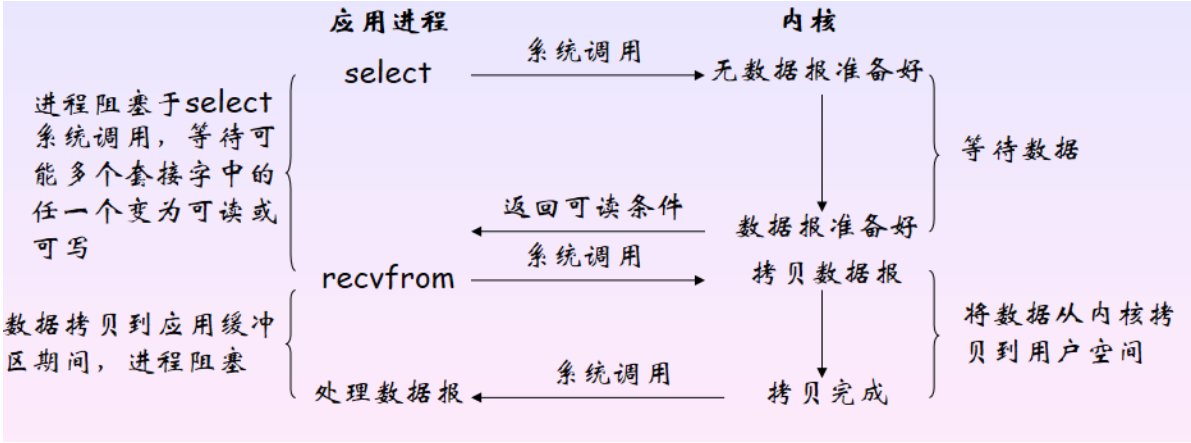
非阻塞I/O



需要设置套接字为非阻塞模式。

使用轮询，浪费CPU资源

I/O复用（重点）



阻塞于系统调用 select函数或poll函数。

可以等待多个I/O，当其中有一个可读或可写时便继续执行。

select函数

等待多个事件中的任意一个发生，或等待一定时间

返回准备好描述字的数量。

```
int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptset,
const struct timeval *timeout);
//maxfd：指等待集合中所有描述符的范围，即所有描述符的最大值加1
//readset：存放检测“读”条件的描述符集合
//writeset：存放检测“写”条件的描述符集合
//exceptset：存放检测“异常”条件的描述符集合。
//若是对上面的某个条件不感兴趣，将其设置为空指针
//这些参数传入时指定要关注哪些描述符，返回时返回关注的哪些已经准备好。
//timeout：指定等待时间，结构可以设置秒与毫秒  空指针为永远等待，0为轮询
```

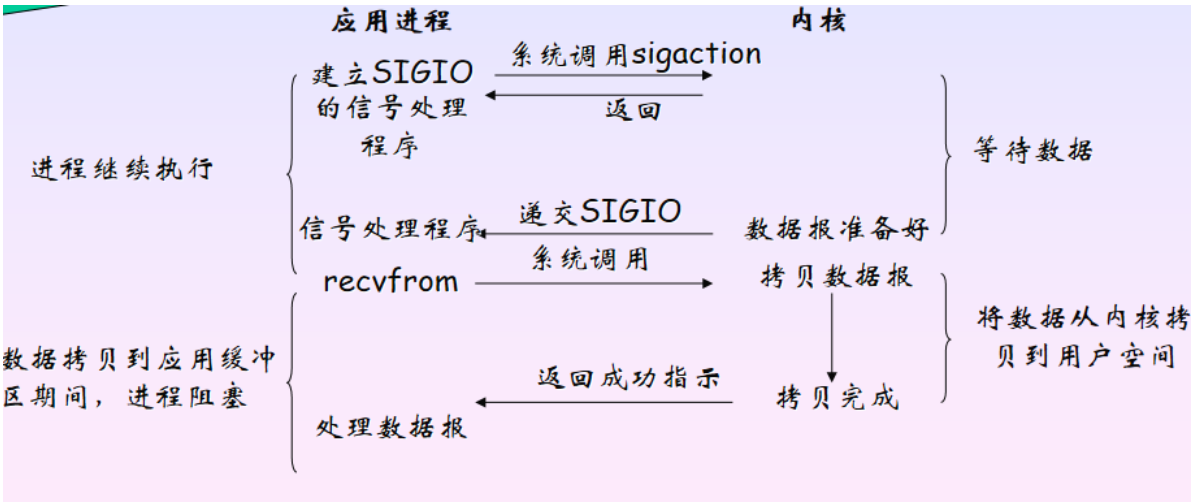
对描述符集合的操作

```
void FD_ZERO(fd_set  *fdset);                                  //清空集合，将所有位设为0
void FD_SET(int fd, fd_set  *fdset);                           //将fd位设为1 在select需要检查该位对应的描述符
void FD_CLR(int fd, fd_set  *fdset);                           //将fd位设为0 不再检查该位对应的描述符
int FD_ISSET(int fd, fd_set  *fdset);                          //检测是否需要检查fd位对应的描述符，即检测该位是否为1
//fdset：指向描述符集合的指针
//fd 对应的描述符，在套接字编程中一般为套接字描述符
//在传入select时，1代表我们关注对应的描述符，传出时，1代表其已经准备好。
```

每次调用select前都必须对等待描述字集合完成初始化和设置工作

代码：

信号驱动I/O（非重点）

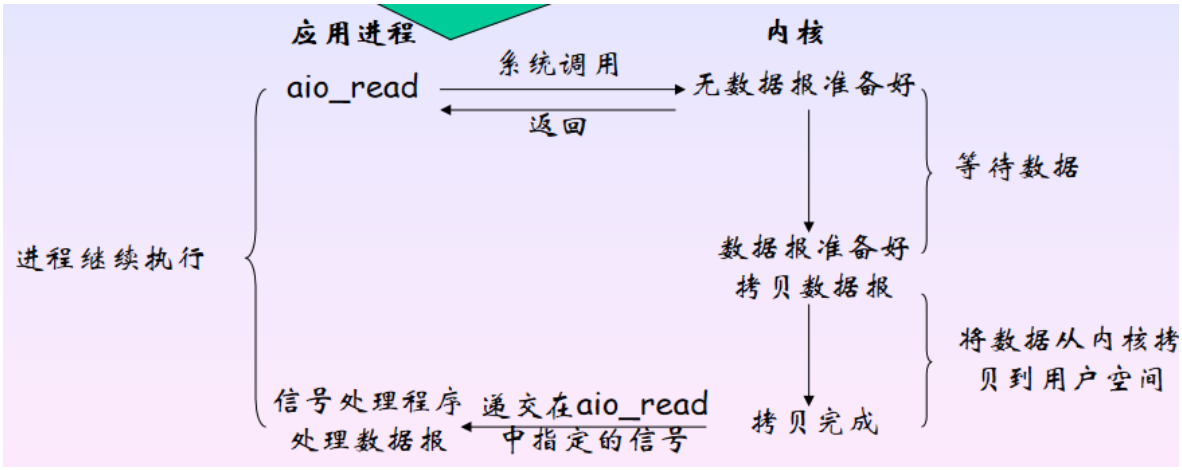


在套接字准备好时通过信号通知进程进行读取。

由信号处理程序来读取数据，或通知主循环来读取数据。

需要允许socket进行信号驱动I/O，并通过系统调用sigaction安装一个信号处理程序，此系统调用立即返回。

异步I/O（非重点）



与信号驱动I/O的区别在于，异步I/O通知何时读取完，信号驱动I/O通知何时可以读取。

代码示例

socket函数(以IPv4 TCP为例子)

```
int sockfd;
   //crteate socket
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) = = -1)
{
    //handle exception

}
```

bind函数

```
    setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));//注意出现的位置，一定在是bind函数之前
    bzero(&addr,sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(port);

    if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) == -1)
    {
        /* 错误处理 */
    }
```

connect函数

```
int sockfd;
struct sockaddr_in server;
……
bzero(&server, sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(1234);
server.sin_addr.s_addr= inet_addr("127.0.0.1");
if (connect(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1)
{
     //handle exception
……
}
```

accept函数

```
struct sockaddr_in  servaddr, cliaddr;
    socklen_t len;
    int listenfd, connfd;
    …
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &len);
    if (connfd == -1)    {
        /* 出错处理 */
    }
    printf(" connection from %s",
        inet_ntop(AF_INET, cliaddr.sin_addr, buff, sizeof(buf))
          );
    …
```

TCP迭代服务器模板

```
int main(void)
{
```

```
    //套接字创建
    int sockfd,connect_sock;

    if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)   {

        perror(“create socket failed.”);
        exit(-1);
    }
    //绑定地址与初始化
    int port=1234;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));//注意出现的位
置，一定在是bind函数之前
    bzero(&addr,sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) == -1)
    {
        /* 错误处理 */
    }
    //转变为监听状态
    int BACKLOG = 5;
    listen(sockfd,BACKLOG);//此处也要进行错误判断
    //监听
    while(1){
           if((connect_sock=accept(sockfd,NULL,NULL))==-1)  {
                perror(“Accept error.”);
                exit(-1);
          }
            //进行读写操作，send recv
            //读写完成
            close(connect_sock);
           }
    close(sockfd);
}
```

TCP迭代服务器实例

```
#define PORT    1234
#define BACKLOG 1
int main(void)
{
        int                     listenfd, connectfd;
        struct sockaddr_in  server;
        struct sockaddr_in client;
        int                     sin_size;
        //创建套接字
        if((listenfd=socket(AF_INET, SOCK_STREAM, 0))==-1)
        {
                perror("Create socket failed");
                exit(-1);
        }
```

```
        //初始化地址与套接字 进行绑定
        int opt =1;
        setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
        bzero(&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(listenfd, (struct sockaddr *)&server, sizeof(struct
sockaddr))==-1)
        {
                perror("Bind error");
                exit(-1);
        }
        //设置套接字为监听模式
        if (listen(listenfd, BACKLOG) == -1)
       {
                perror("listen error");
                exit(-1);
        }
        //接受连接-通信循环
        sin_size = sizeof(struct sockaddr_in);
        while(1)
        {
            if ((connectfd = accept(listenfd, (struct sockaddr *)&client,
&sin_size)) == -1)
                {
                     perror("accept error");
                     exit(-1);
                }
             printf("you get a connection from %s\n",
inet_ntoa(client.sin_addr));
             send(connectfd,"welcome to my server\n",22, 0);
             close(connectfd);
        }
        close(listenfd);
}
```

TCP客户端模板

```
int main(void)
{
    //创建套接字
    int sockfd;
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))=-1)
    {
        perror(“Create socket failed.”);
        exit(-1);
    }
    //初始化并链接
    struct sockaddr_in server;

    bzero(&server, sizeof(server));
```

```
    server.sin_family = AF_INET;
    server.sin_port = htons(0);
    server.sin_addr.s_addr= inet_addr(INADDR_ANY);
    if (connect(sockfd, (struct sockaddr *)&server, sizeof(server)) == -1)
    {
     //handle exception
    }

    //进行读写操作，send recv
    //读写完成
    close(sockfd);
}
```

TCP客户端实例

```
#define PORT            1234
#define MAXDATASIZE     100
int main(int argc, char *argv[])
{
        int fd, numbytes;
        char        buf[MAXDATASIZE];
        struct  hostent *        he;
        struct sockaddr_in server;
        //通过命令行参数读取IP
        if (argc != 2)
        {
                printf("Usage: %s  <IP address>\n", argv[0]);
                exit(-1);
        }
        if ((he = gethostbyname(argv[1])) == NULL)
        {
                perror("gethostbyname error.");
                exit(1);
        }
        //创建套接字
        if ((fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
                perror("Create socket failed.");
                exit(1);
        }
        //初始化并链接
        bzero(&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr = *((struct in_addr *) he->h_addr);
        if (connect(fd, (struct sockaddr *)&server, sizeof(struct sockaddr)) ==
-1)
        {
                perror("connect failed.");
                exit(1);
        }
        //接收数据
        if( ((numbytes = recv(fd, buf, MAXDATASIZE, 0)) == -1))
        {
```

```
                perror("recv error.");
                exit(1);
        }
        buf[numbytes] ='\0';
        printf("Server Message: %s\n",buf);
        close(fd);
}
```

UDP服务器实例

```
#define PORT    1234
#define MAXDATASIZE     100
int main(void)
{
        int                     sockfd;
        struct sockaddr_in      server, client;
        int                     sin_size, num;
        char                    msg[MAXDATASIZE];
        //创建套接字
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)            {
                perror("Create socket failed");
                exit(1);
        }
        //初始化并绑定
        bzero(&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(sockfd, (struct sockaddr *)&server, sizeof(struct sockaddr)) ==
-1) {
                perror("Bind error.");
                exit(1);
        }
        //消息接收循环
        sin_size = sizeof(struct sockaddr_in);
        while(1) {
                num = recvfrom(sockfd, msg, MAXDATASIZE, 0, (struct sockaddr
*)&client, &sin_size);
                if (num < 0)  {
                        perror("recvfrom error.");
                        exit(1);
                }
                msg[num] ='\0';
                printf("You got a message (%s) from %s\n",
msg,inet_ntoa(client.sin_addr));
                sendto(sockfd,"Welcome",strlen("welcome"), 0, (struct sockaddr
*)&client, sin_size);
                if (!strcmp(msg,"quit"))  break;
        }
        close(sockfd);
}
```

UDP客户端示例

```
#define PORT            1234
#define MAXDATASIZE     100

int main(int argc, char *argv[])
{
        int fd, numbytes;
        char    buf[MAXDATASIZE];
        struct hostent *he;
        struct sockaddr_in server, reply;
        //通过命令行参数读取IP
        if (argc != 3)
        {
                printf("Usage: %s  <IP address>  <Message>\n", argv[0]);
                exit(1);
        }

        if ((he = gethostbyname(argv[1])) == NULL) {
                perror("gethostbyname error\n");
                exit(1);
        }
        //创建套接字
        if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
                perror("Create socket failed");
                exit(1);
        }
        //初始化地址
        bzero(&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr = *((struct in_addr *) he->h_addr);
        sendto(fd, argv[2], strlen(argv[2]), 0, (struct sockaddr *)&server,
sizeof(struct sockaddr));
        //读取消息
        int len=sizeof(struct sockaddr_in);
        while(1)  {
        numbytes = recvfrom(fd,buf, MAXDATASIZE, 0, (struct sockaddr *)&reply,
&len);
        if (numbytes ==-1)  {
                        perror("recvfrom error.");
                        exit(1);
                }
        //判断信息是否来自服务器
        if (len != sizeof(struct sockaddr_in) || memcmp((const void *)&server,
(const void *)&reply, len)!= 0)
        {
                        printf("receive message from other server.\n");
                        continue;
                }
                buf[numbytes] ='\0';
                printf("Server Message: %s\n",buf);
                break;
        } /* while(1) */
        close(fd);
}
```

TCP并发（多线程）服务器模板

```
int main(void)
{
    int     listenfd, connfd;
    pid_t      pid;
    int BACKLOG = 5;
    //创建套接字
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)  {
        perror("Create socket failed.");
        exit(1);
    }
    //绑定
    bind(listenfd, …);
    //监听
    listen(listenfd, BACKLOG);
    //消息接收循环
    while(1) {
            if ((connfd = accept(sockfd, NULL, NULL)) == -1)  {
                perror("Accept error.");
                 exit(1);
            }
            if((pid = fork() ) > 0){//创建子进程
            //若为父进程
                close(connfd);
                continue;
            }
            else if (pid == 0){
            //若为子进程
                close(lisetenfd);
                //通信
                exit(0);
            }
            else{
                printf("fork error\n");
                exit(1);
            }
        }
    }
```

TCP并发（多线程）服务器实例

```
#include <stdio.h>
……
#define  PORT   1234
#define  BACKLOG    10
#define MAXDATASIZE 1000
void process_cli(int connectfd, struct sockaddr_in client);
int main(void)
{
    int     listenfd, connectfd;
    pid_t   pid;
    struct sockaddr_int server, client;
```

```
    int     sin_size;

    //创建套接字
    ……
        if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            perror("Create socket failed");
            exit(-1);
        }
    //初始化并绑定
    int opt = SO_REUSEADDR;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    bzero(&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    server.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(listenfd, (struct sockaddr*)&server, serrver)) == -1)
    {
        perror("Bind error");
        exit(-1);
       }
    //设置套接字为监听模式
    if (listen(listenfd, BACKLOG) == -1)
    {
        perror("listen error");
        exit(-1);
    }
    //接收连接并处理
    sin_size = sizeof(struct sockaddr_in);

    while (1) {

        if ((connectfd = accept(listenfd, (struct sockaddr*)&client, &sin_size))
== -1) {
            perror("accept error");
            exit(-1);
        }
        if ((pid = fork()) > 0) {
            //若为父进程
            close(connectfd);
            continue;
        }
        else if (pid == 0) {
            //若为子进程
            close(listenfd);
            process_cli(connectfd, client);
            exit(0);
        }
        else {
            printf("fork error\n");
            exit(0);
        }
    }/*while()*/
```

```
    close(listenfd);   /* close listenfd */
}
void process_cli(int connectfd, struct sockaddr_in client) {//通信函数
    int     num;
    char        recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
cli_name[MAXDATASIZE];
    printf("You got a connection from % s.\n", inet_ntoa(client.sin_addr));
    num = recv(connectfd, cli_name, MAXDATASIZE, 0);
    if (num == 0) {
        close(connectfd);
        printf("cllient disconnected.\n");
        return;
    }
    cli_name[num] = '\0';
    printf("Client name is % s.\n", cli_name); :
    while (num = recv(connectfd, recvbuf, MAXDATASIZE, 0) {//直到对方close才退出循环
        recvbuf[num] = '\0';
        printf("Received client(% s) message: % s\n", cli_name, recvbuf);
        for (int i = 0; i < num; i++)
            sendbuf[i] = recvbuf[num - i - 1];
        sendbuf[i] = '\0';
        send(connectfd, sendbuf, strlen(sendbuf), 0);
    }
    close(connectfd);
}
```

TCP并发（多线程）服务器模板

```
void *start_routine( void *arg);
int main(void)  {
    int listenfd, connfd;
    pthread_t   tid;
    type        arg;
    //创建套接字
    ……
    //初始化地址并绑定套接字
    ……
    //设置为监听模式
    ……
    while(1) {
        //接收到连接请求
        if ((pthread_create(&tid, NULL, start_routine, (void *)&arg))
            /* handle exception */
        ……}
    ……}
```

TCP并发（多线程）服务器实例

```
#include <stdio.h>
……
#define  PORT   1234
```

```
#define  BACKLOG    2
#define MAXDATASIZE 1000
void process_cli(int connectfd, sockaddr_in client);
void* start_routine(void* arg);
struct ARG {
    int         connfd;
    sockaddr_in     client;
};
int main(void)
{
    int listenfd, connectfd;
    pthread_t   tid;
    ARG* arg;
    struct sockaddr_in     server, client;
    int sin_size;
    //创建，绑定，监听在此处都略过
        sin_size = sizeof(struct sockaddr_in);
    while (1) {
        if ((connectfd = accept(listenfd, (struct sockaddr*)&client,
            &sin_size)) == -1)
            /* handle error */
            arg = new ARG;//相当于malloc，重点在于分配空间
        arg->connfd = connectfd;//赋值
        memcpy((void*)&arg->client, &client, sizeof(client));//赋值，注意值的性质导
致必须用这个函数
        if (pthread_create(&tid, NULL, start_routine, (void*)arg)
            /* handle erroe */
    }
    close(listendfd);
}
void process_cli(int connectfd, sockaddr_in client) {
    …
}
void* start_routine(void* arg) {
    ARG* info;
    info = (ARG*)arg;
    process_cli(info->connfd, info->client);
    delete arg;
    pthread_exit(NULL);
}
```

线程安全性函数

```
static pthread_key_t        key;
static pthread_once_t   once = PTHRAD_ONCE_INIT;
typedef struct DATA_THR {
    int     index;
}
static void destructor (void *ptr) {//析构函数
    free(ptr);
}
static void getkey_once(void) {//创建关键字
    pthread_key_create (&key, destructor);
```

```
}
void savedata(char *recvbuf, int len, char *cli_data) {
    DATA_THR * data;

    pthread_once(&once, getkey_once);
    if ((data = (DATA_THR *)pthread_getspecific ( key )) == NULL)  {//判断并进行初始化
        data = (DATA_THR *) calloc (1, sizeof(DATA_THR));
        pthread_setspecific (key, data);
        data -> index = 0;
    }
    for (int i=0; i< len -1; i++)//保存数据
        cli_data[data -> index++] = recv[i];
        cli_data[data -> index] = '\0';
}
```

I/O复用模板

```
//前面掠过

while (1)
{
    //进行初始化
    FD_ZERO(&infds);
    FD_SET(fileno(stdin), &infds);//取文件描述符
    FD_SET(sockfd, &infds);
    maxfd = max(fileno(stdin), sockfd) + 1;
    if (select(maxfd, &infds, NULL, NULL, NULL) == -1)
    {
        fprintf(stderr, "select error in udptalk.c\n");
        exit(3);
    }
    if (FD_ISSET(sockfd, &infds))//判断是否是通信套接字
    {
        n = read(sockfd, msg, BUFLEN);
        if ((n == -1) || (n == 0)) {
            printf("peer closed\n");
            exit(0);
        }
        else {
            msg[n] = 0;
            printf("peer:%s", msg);
        }
    }
    if (FD_ISSET(fileno(stdin), &infds))//判断是否是标准输入
    {
        if (fgets(msg, BUFLEN, stdin) == NULL) {
            printf("talk over!\n");
            exit(0);
        }
        write(sockfd, msg, strlen(msg));
        printf("sent:%s", msg);
    }
```

```
}
```

I/O复用实例

```
#include <stdio.h>
…
#define PORT    1234
#define BACKLOG 5
#define MAXDATASIZE 1000
tpedef struct CLIENT {
    int         fd;
    char* name;
    struct  sockaddr_in addr;
    char* data;
};
void process_cli(CLIENT* client, char* recvbuf, int len);
void savedata(char* recvbuf, int len, char* data);
int main(void) {
    int i, maxi, maxfd, sockfd;
    int nready;
    ssize_t n;
    fd_set rset, allset;
    int listenfd, connectfd;
    struct sockaddr_in  server;
    CLIENT  client[FD_SETSIZE];
    char recvbuf[MAXDATASIZE];
    int sin_size;
        //创建，绑定，监听在此处都略过
        /* Create TCP socket */          ……
        /* Bind address */  ……
        /* Listen socket listenfd */     ……


        //初始化描述符集合
    sin_size = sizeof(struct sockaddr_in);
    maxfd = listenfd;
    maxi = -1;
    for (i = 0; i < FD_SETSIZE; i++)
        client[i].fd = -1;//有意义的描述符非负整数
    FD_ZERO(&allset);//归零
    FD_SET(listenfd, &allset);//关注监听描述符
    while (1) {
        struct sockaddr_int addr;
        rset = allset;//保存一个副本
        nready = select(maxfd + 1, &rset, NULL, NULL, NULL);//检查是否有可用的IO
        if (FD_ISSET(lisetenfd, &rset)) {//判断是否为监听套接字
            if ((connectfd = accept(listenfd, (struct sockaddr&)&addr,
&sin_size)) == -1) {
                perror(“accept error.”);
                continue;
            }
            for (i = 0; i < FD_SETSIZE; i++)//监听成功，对客户端数据进行保存
                if (client[i].fd < 0) {//找到仍未赋值的数组元素
```

```
                    //保存数据
                    clinet[i].fd = connectfd;
                    client[i].name = new char[MAXDATASIZE];
                    client[i].addr = addr;
                    client[i].data = new char[MAXDATASIZE];
                    client[i].name[0] = '\0';
                    client[i].data[0] = '\0';
                    printf(" You got a connect from % s.",
inet_ntoa(client[i].addr.sin_addr));
                    break;
                }
                //判断是否为满
                if (i == FD_SETSIZE)    printf("too many cllients.\n");
                //关注该客户端对应的套接字
                FD_SET(connectfd, &allset);
                //进行对应的赋值
                if (connectfd > maxfd)   maxfd = connectfd;
                if (i > maxi)   maxi = i;//保存目前最大的已连接套接字
                if (--nready < = 0) continue;//判断是否仍有可用的套接字未处理
            }  /* if (FD_ISSET (listenfd… */
            for (i = 0; i <= maxi; i++) {
                if ((sockfd = client[i].fd) < 0)    continue;//过滤掉未赋值的
                if (FD_ISSET(sockfd, &rset)) {//判断是否可用
                    if ((n = recv(sockfd, recvbuf, MAXDATASIZE, 0)) == 0) {//判断
客户端是否close
                        close(sockfd);
                        printf("Client(% s) closed connection.User's data : %
s\n", client[i].name, client[i].data);
                        FD_CLR(sockfd, &allset);//不再关注该套接字
                        client[i].fd = -1;//恢复至未赋值状态
                        delete client[i].name;//相当于free
                        delete client[i].data;
                    }
                    else
                        process_cli(&client[i], recvbuf, n);
                    if (--nready <= 0)  break;
                }  /* if (FD_ISSET(sockfd, &rset))  */
            } /* for(i = 0; i <= maxi; i++)   */
        }  /* while(1); */
    close(listenfd);
}
```